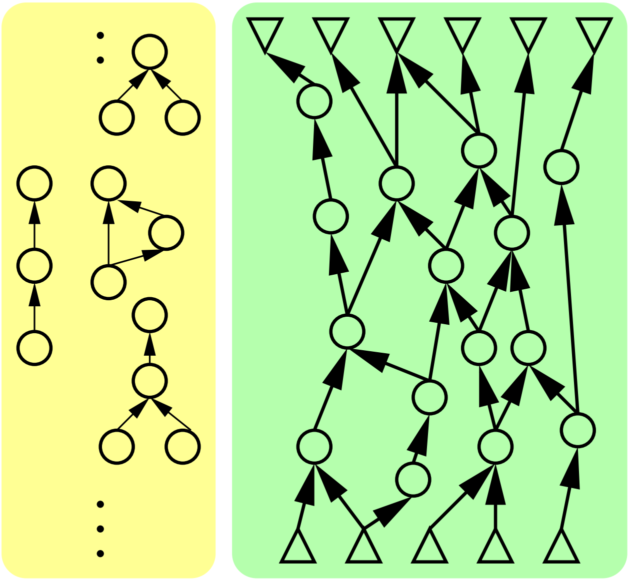
Flattening of Basic Blocks for Preaccumulation

J. Utke

given a sequence of statements: "How can I get and use a DAG for preaccumulation?"

- where is this needed
- why doesn't somebody else do it for me
- how is it done
- what makes it ambiguous
- which choices do we have

ACTS project and implementation in OpenAD



where do we need it?

cross country elimination

- vertex/edge/face elimination
- scope beyond single statement
- pre-accumulate Jacobian entries j_{yx}
- and propagate forward `saxpy(`j_{yx}`,` $\dot{x}$`,` $\dot{y}$`)`
- or stack them, then reverse through the stack
- storing cheaper than recomputes
- fewer Jacobian entries than intermediate values, etc.
- or scarcity preserving elimination
- concentrate on basic blocks (loop body, low level routine with straight line code)
- need a DAG
- note, can extend beyond basic block scope (resolve side effects)

Shouldn't the compiler do it?

YES!

- code optimization
- register allocation
- code generation, etc.

BUT, we do high level source to source transformation

- transformation starts after parsing/canonicalizing/filtering
- compiler code optimization happens at a later stage
- $\rightarrow$ not available
- unless we go to low level transformation or elevate compiler optimization

We have to do it ourselves.

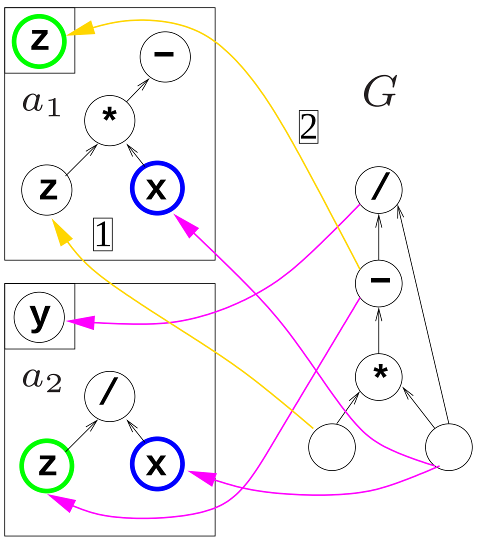
We can do what we want ☺.

simple flattening

simple case:

a_1 : `z=-(z*x)`

a_2 : `y=z/x`



- sequence of assignments in a basic block
- front end provides rhs expressions as graphs
- algorithm to flatten them into a single graph G:
  - iterate through all assignments in sequence order
  - replicate the rhs$_s$ in G
  - identify variables
    - within a rhs, if the front-end hasn't already done that (size↓)
    - across rhs$_s$ (size↓)
    - between rhs$_s$ and lhs$_s$, preserves semantics!
  - track the most recent assignment to a v
- variable identification is easy for plain scalar values (syntactic equivalence)
- otherwise through (flow-sensitive) must alias analysis, i.e. identification by unique (virtual) address.

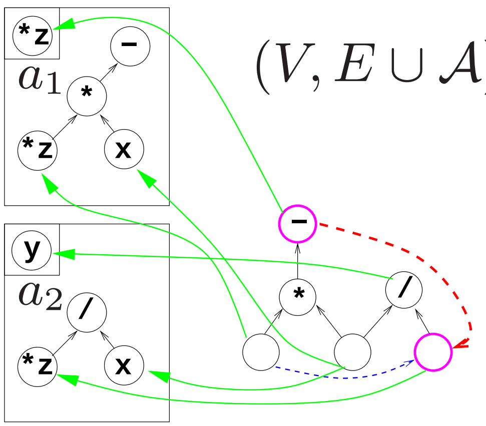
why (virtual) addresses?

we have vectors, pointers, etc. $\rightarrow$ likely only have may alias

with aliasing:

a_1 : `*z=-(*z*x)`

a_2 : `y=*z/x`

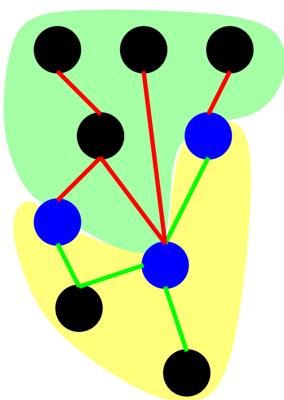


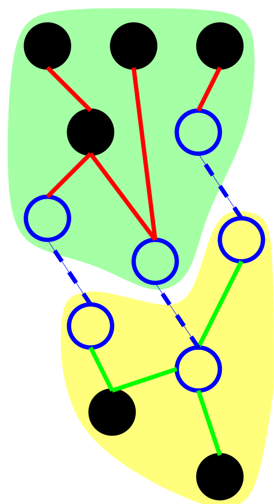
$(V, E \cup \mathcal{A})$

- simple algorithm creates new vertex if not uniquely identifiable
- G is incomplete (missing edges)
- may-aliases establish *virtual* edges $\in \mathcal{A}$ indicating possible identification (only want references of rhs vertex to preceding lhss)
- $G' = (V, E \cup \mathcal{A})$ is a set of possible dags, only one element preserves semantics
- resolve ambiguities by splitting

edge subgraphs

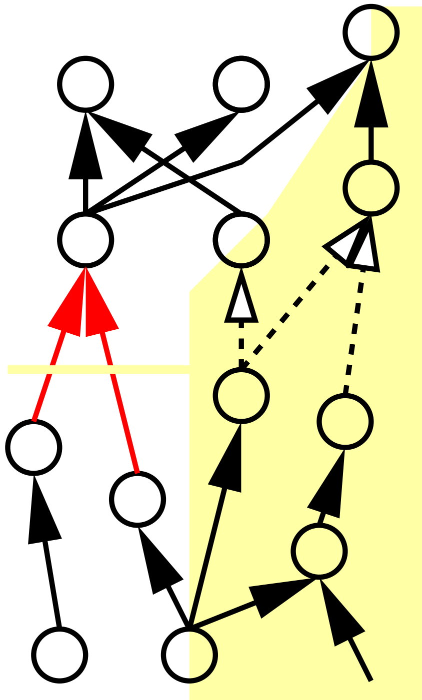
edge split:

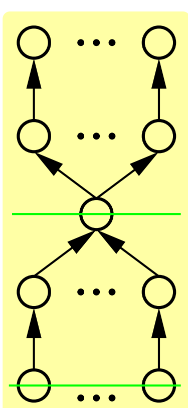




- define *edge* subgraph $G_s = (V_s, E_s)$ of $G = (V, E)$ with $V_s \subseteq V$ and $E_s \subseteq E$ such that if $(v, w) \in E_s$ then $v, w \in V_s$ and if $(t, u), (v, w) \in E_s \wedge (u, v) \in E$ then $(u, v) \in E_s$
- define *split* of G into edge subgraphs $G_i = (V_i, E_i)$ such that $E = \bigcup E_i \wedge E_i \cap E_j = \emptyset$ (reverse of flatten; example: E_1, E_2)
- split $(V, E \cup \mathcal{A})$ into edge subgraphs G_i that
  - have $\mathcal{A}_i = \emptyset$, i.e. locally unambiguous dependency information
  - are (partially) ordered with '$\prec$' such that $\forall (v, w) \in \mathcal{A} : v \in G_j$ then $w \in G_k, G_j \prec G_k$ and
  - $\forall (t, u) \in \mathcal{A} : u \in G_j$ then $t \in G_i, G_i \prec G_j$

split choices





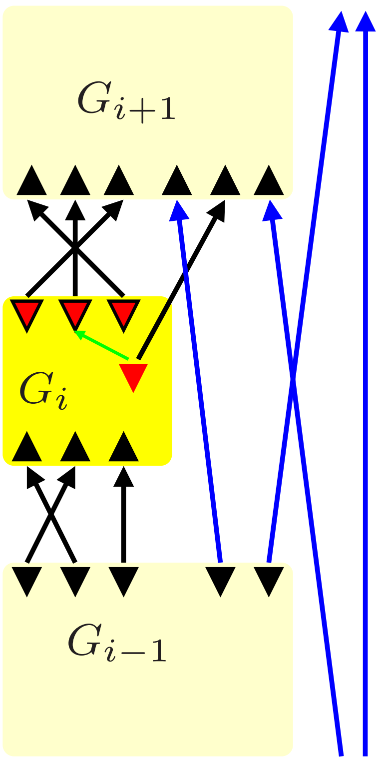
splitting

- criteria define a minimal number of splits
- $\exists$ *movable* edges $\rightarrow$ split itself isn't fully defined

in the example:

- $\mathcal{A} = \{(v,w),(v',w)\}$
- movable edges (t,u) if $\nexists\ P_{u,v}, P_{u,v'}, P_{w,t}$
- space for optimization, e.g. $\sum n_i m_i p + ops(G_i)$; gains ?
- array ops
- minimal cost doesn't imply minimal split count (scarcity)

in practice I



w/o array ops

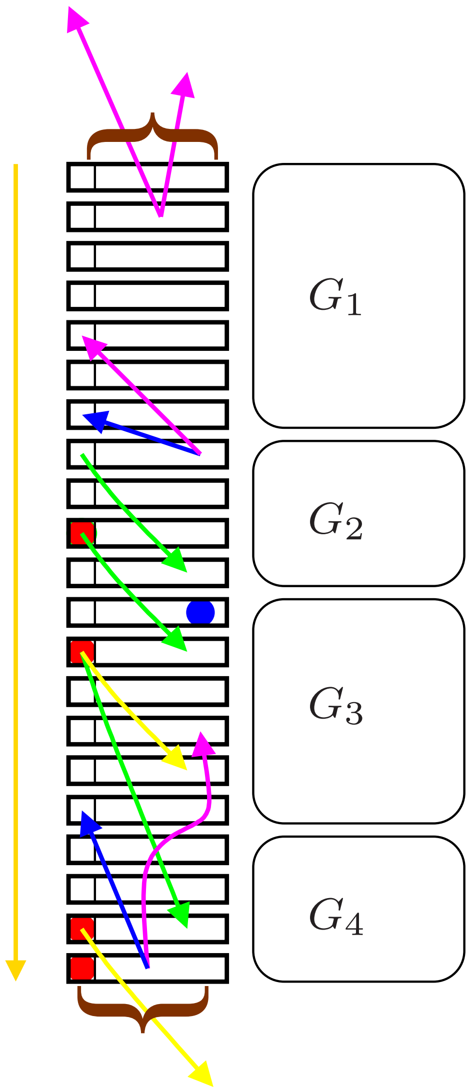
- pick splits along assignment borders
- preserves semantics

determining Jacobian entries ?

- sequence of G_i leads to sequence of Jacobians $\boldsymbol{J}_i$, $\boldsymbol{J} = \prod \boldsymbol{J}_i$
- $\boldsymbol{J}_i = \boldsymbol{P}_i^{(r)} \boldsymbol{C}_i \boldsymbol{P}_i^{(c)}$ where $\boldsymbol{C}_i = \begin{bmatrix} \boldsymbol{J}_{G_i} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix}$
- $\boldsymbol{I} \in \mathbb{R}^{s_i \times s_i}, s_i = |\{(v,w) \in \mathcal{I}, v \in V_j, w \in V_k \, j < i < k\}|$
- $\mathcal{I}$ are identities between vertices, $\boldsymbol{P}_i^{(r)}, \boldsymbol{P}_i^{(c)}$ permute rows / columns
- $\mathcal{I}, \boldsymbol{P}_i^{(r)}, \boldsymbol{P}_i^{(c)}$ only known at runtime
- inputs are minimal vertices in G_i (easy)
- $\boxed{\text{maximal vertices}} \subseteq out(G_i) \subseteq$ final lhss
- assuming $out(G_i) \equiv$final lhss complicates the graph

  $\rightarrow$ basic block elimination looses potential

in practice II



alias++ :

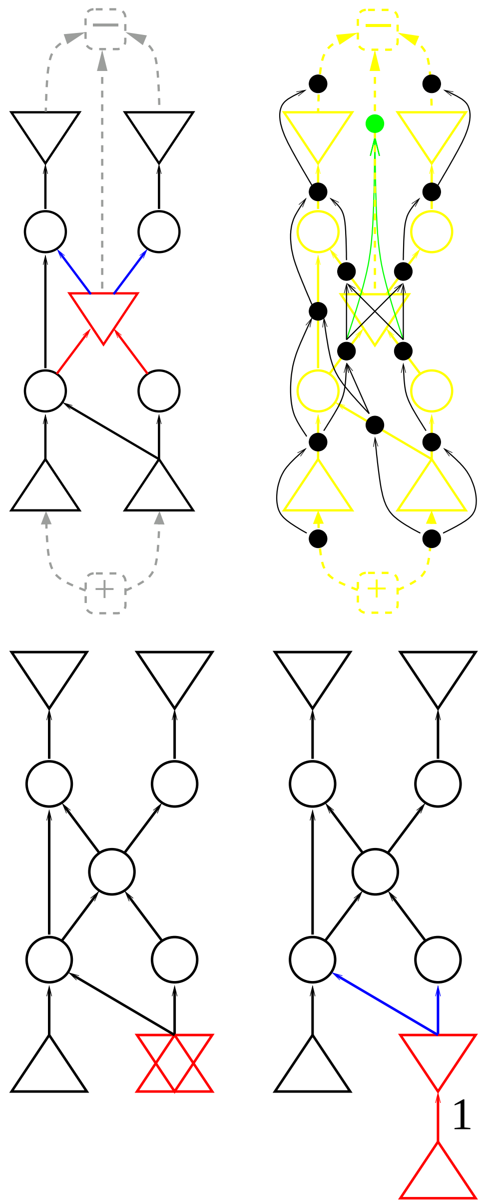
- *scalar replacement* of pointer/array derefs & alias analysis block local
- $\equiv$ $\boxed{ud}$-*chain* information (possible definitions for vertex v)
- we need:
  - reference $r(ud_v)$ to most recent definition (i.e. assignment) in this basic block
  - determine if definitions are
    - ambiguous (inside,both sides, outside), or
    - unique (inside, outside)

Jacobian rows:

- $\boxed{du}$-*chain* information (possible use of this lhs v)
  $r(du_v)$ referencing the last use in this basic block, if $\exists$
- v output in G_i ?: $\exists r(du_v) \wedge r(du_v) \notin V_i \vee \nexists\, r(du_v)$

missing information forces splits/Jacobian rows (conservative default)
can degenerate to statement level pre-accumulation

weird cases



non-maximal dependent :

- vertex elimination requires vertex/edge duplication
- edge elimination has constraints to back elimination steps
- face elimination not affected

$y = x$:

- standalone vertex → filtered out
- independent/dependent merge → insert trivial edge → constant folding

subroutine calls, extending scope

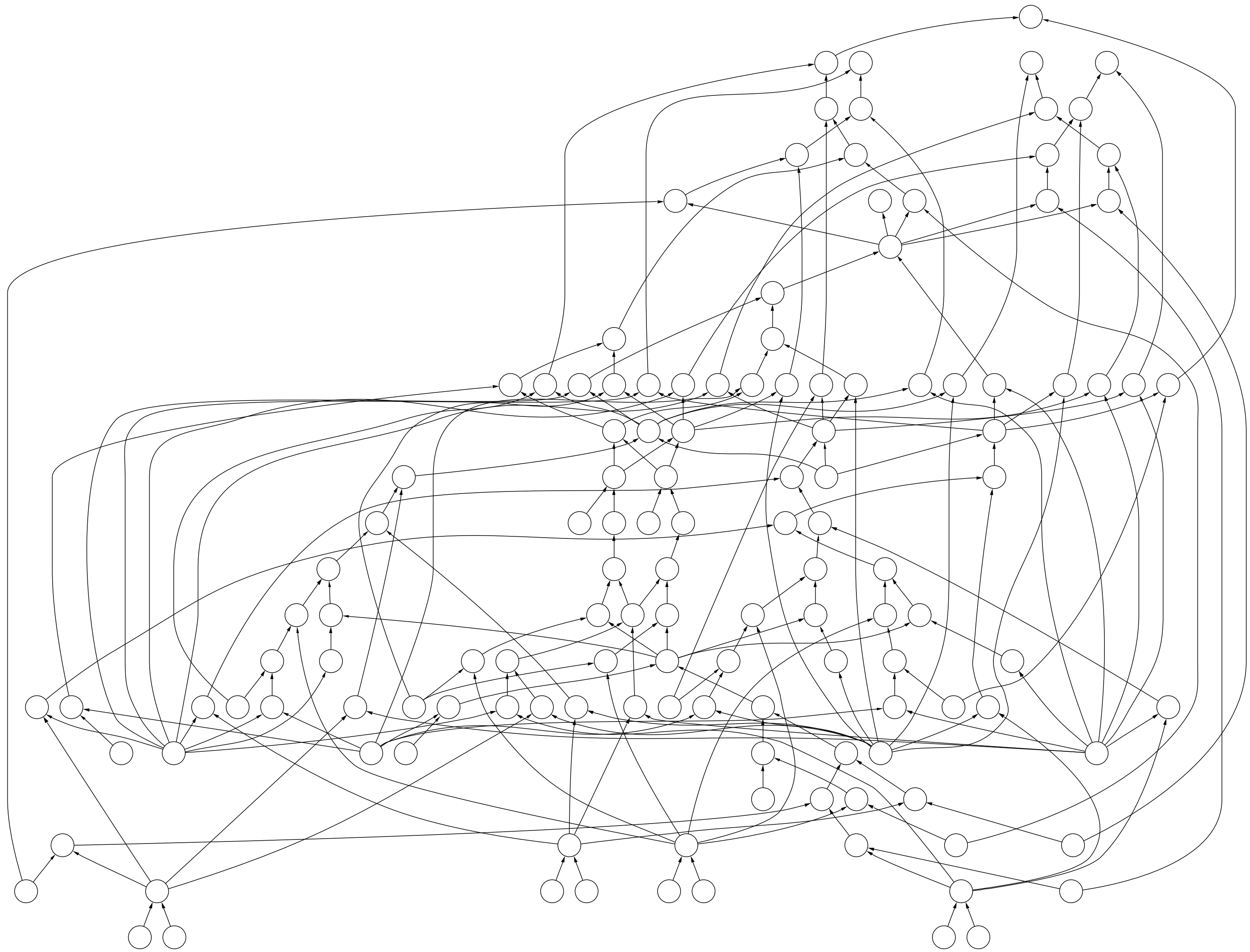
SR call treatment:

- side effect free?
- user defined black box?

extending the scope:

- no restrictions on du/ud chains
- looking at loops !
- handling branches and inlining ?

example, 131 vertices



Summary

implemented and used in OpenAD

- front-end parses code and provides intermediate representation
- OpenAnalysis component provides alias, ud/du chain included in IR
- algorithm builds DAGs
- heuristic approximates optimal elimination sequence
- algorithm generates partial calculation and elimination steps code to IR
- algorithm adds saxpy calls to IR
- front-end unparses IR $\rightarrow$ ad code

Higher level approaches depend on coding style